Mapping Polymorphism

Ryan Wisnesky
Harvard University
ryan@cs.harvard.edu

Mauricio A. Hernández, Lucian Popa
IBM Almaden Research Center
{mauricio, lucian}@almaden.ibm.com

ABSTRACT

We examine schema mappings from a type-theoretic perspective and aim to facilitate and formalize the reuse of mappings. Starting with the mapping language of Clio, we present a type-checking algorithm such that typable mappings are necessarily satisfiable. We add type variables to the schema language and present a theory of polymorphism, including a sound and complete type inference algorithm and a semantic notion of a principal type of a mapping. Principal types, which intuitively correspond to the minimum amount of schema structure required by the mappings, have an important application for mapping reuse. Concretely, we show that mappings can be reused, with the same semantics, on any schemas as long as these schemas are expansions (i.e., subtypes) of the principal types.

Categories and Subject Descriptors

H.2.5 [**Heterogeneous Databases**]: Data Translation; H.2.3 [**Languages**]: Query Languages; D.3.3 [**Language Constructs and Features**]: Polymorphism

General Terms

Languages, Theory

Keywords

Schema Mapping, Nested Relational Model, Clio

1. INTRODUCTION

Data exchange is the process of transforming data instances of one or more source schemas into instances of a target schema. Much research about data exchange describes the process using a high-level and declarative formalism called *schema mappings* [25, 27, 10]. Schema mappings (or *mappings* in short) are logical assertions that express constraints between two or more data sources. In particular, schema mappings are used to capture how data conforming to a source schema corresponds to data conforming to a target schema.

The development of schema mappings and their application to data exchange was pioneered by the Clio project (see [9] for a recent retrospective on Clio). Clio concentrated on generating schema mappings from even higher-level constructs like *correspondences* (or matches) between schema elements, and then converting the generated mapping into queries or programs that capture the trasformation semantics of the mappings. In particular, Clio developed a number of query generators to convert mappings into various XML or relational transformation languages (e.g., SQL, XQuery, XSLT, SQL/XML).

Figure 1 shows Clio in action. The source-to-target lines (also called *correspondences*) are entered by the user and indicate how atomic elements in the source relate to atomic elements in the target. The lines on the left- and right-hand-side of the schemas represent the foreign-key constraints that are given with the schemas. Given such schemas with constraints, a set of correspondences, and also through a fair amount of user interaction, Clio generates a set of schema mappings that best represent the "intention" behind the visual specification of the mapping. In this example, the five correspondences shown in the figure would be translated into a mapping that requires that each record in the source set gradEnroll be split over three target sets: eval, Student and course. (In the figure, the elements that have the [0,*] suffix denote sets.)

One important and desirable feature of the generated mapping is that it preserves data associations. For the above example, this means that the individual values of the input record (i.e., sid, name, cid, grade, file) will remain connected in the target (by exploiting the foreign key constraints and the schema structure). We shall give the exact mapping in Section 2, in a language that follows the source-to-target tuple-generating dependencies of [10].

IBM [19], Microsoft [5], Oracle [6], and others are building an ecosystem of tools around schema mappings, where mappings are building blocks for more complex data transformations. Models and semantics of schema mappings for data exchange [10], operations over mappings (e.g., composition [23, 11] and inversion [8, 1]), and the application of such operations to metadata management [2, 4] have been extensively studied within the database and information integration community over the past decade.

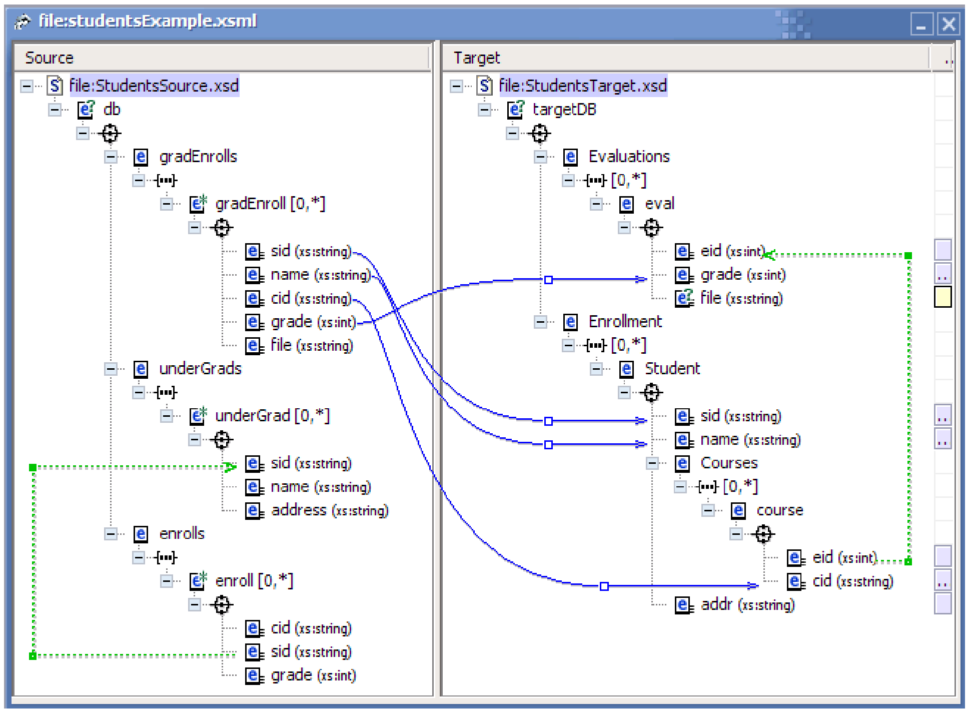

Figure 1: A schema mapping in Clio

However, the problem of **reusing schema mappings** has been largely ignored. Schema mappings are, in general, neither parametric nor modular. That is, mappings are static expressions over a source and a target schema and can not be easily reused over other schemas. Large schemas often combine and reuse existing type definitions to define larger and more complex types. For example, a *Person* type can be defined and used whenever the attributes of a *Person* are needed in the schema definition. Current mapping languages cannot take advantage of this modularity and cannot express a mapping between expansions of these types (without knowing the concrete expansions). Ideally, we would prefer to create generic mappings between source and target types (e.g., between the source *Person* type and a target *Student* type) and then reuse those mapping within larger mappings. Similarly to regular programming languages, users should be able to build libraries of reusable mappings between common data types. Mappings in these libraries can be reused on larger data transformation scenarios, as building blocks for more complex mappings.

This paper studies how to apply programming language techniques to mapping languages to formally define when and how mappings are reusable. We propose *polymorphism* as a lightweight formal technique for reuse. We use standard type-checking techniques to determine if a mapping applies to a certain context. We further show that mappings have a natural semantic notion of *principal type*, corresponding intuitively to the minimum amount of schema structure required by the mappings. We then show that principal types can be soundly and completely inferred from the structure of the logical assertions alone (i.e. independently of the underlying schemas). In turn, this enables the following mapping reuse technique. Given a mapping M, which is originally defined from a concrete source schema S to a concrete target schema T, we first infer a principal source schema $\hat{S}$ and a principal target schema $\hat{T}$. We can then reuse the same mapping M on any source schema that is an expansion (i.e., subtype) of $\hat{S}$ and any target schema that is an expansion (i.e., subtype) of $\hat{T}$. Moreover, we show in a precise sense that the semantics of M remains invariant during reuse.

This paper is organized as follows. We begin by reviewing schema mappings in Section 2. We then formally define nested schemas and nested data in Section 3. The core mapping language that we focus in this paper is formally defined in Section 4. We then formalize a type-checking algorithm in Section 5. We add type variables to the schema language and present a theory of polymorphism, including a sound and complete type inference algorithm in Section 6, and a semantic notion of principal type in Section 7. All proofs are available in a technical report [32].

2. SCHEMA MAPPINGS: PRELIMINARIES

Several mapping languages have been proposed over the years. In the case of relational schemas, a language popular in the data exchange and data integration community is that of *source-to-target tuple generating dependencies (s-t tgds)* [10] or *(Global-and-Local-As-View) GLAV mappings* [13, 21]. Essentially, these are formulas of the form

$$\forall \mathbf{x}(\phi(\mathbf{x}) \rightarrow \exists \mathbf{y}\, \psi(\mathbf{x}, \mathbf{y}))$$

where ϕ is a conjunction of relational atoms over the source schema, and ψ is a conjunction of relational atoms over the target schema.

The language of s-t tgds was extended to handle hierarchical (XML) schemas in [27]. There, the extended language must account for relations nested inside relations. However, the $\forall\exists$ shape of the dependencies remains the same. An orthogonal extension has been introduced in [14], where in addition to the nesting in the data, we may also have nesting in the mapping formulas themselves (i.e., a $\forall\exists$ type of mapping as above can appear as one of the atoms of ψ in another $\forall\exists$ mapping). It is this type of more flexible, nested mappings that are used in Clio and that we target in this paper.

Although we are using Clio's mapping language [14] as a starting point[1], for most of our results it will be convenient to use a simpler and more uniform mapping syntax, which we introduce later in Section 4. Furthermore, with an eye toward enabling interoperability between mapping systems and conventional programming languages, and unlike [27], we will explicitly represent nested relational data using simple algebraic datatypes.

2.1 Clio mappings: An Overview

The Clio mapping language uses a for - where $\Rightarrow$ exists - where syntax. Intuitively, the for clause binds variables to tuples in the source, and the first (optional) where clause describes the source constraints to be satisfied by these source tuples (e.g., these constraints express filters or join conditions). The exists clause describes the tuples that are expected to exist in the target, and the second where clause describes the target constraints to be satisfied by the target tuples as well as the content of these target tuples in terms of the source tuples.

For our example in Figure 1, the mapping that was informally discussed in Section 1 can be written as:

```
for $g$ in $db$.gradEnrolls.gradEnroll
$\Rightarrow$ exists $s$ in $targetDB$.Enrollment.Student,
          $c$ in $s$.Courses.course,
          $e$ in $targetDB$.Evaluations.eval
    where $s$.sid = $g$.sid $\wedge$ $s$.name = $g$.name $\wedge$
          $c$.cid = $g$.cid $\wedge$ $e$.grade = $g$.grade $\wedge$
          $e$.eid = $c$.eid
```

Note that there may be dependencies between variables, reflecting the nesting in the data. For example, the binding c in s.Courses.course reflects the fact that c must be an element in the set course that is nested under s.Courses, while s itself is an element of the top-level set Student. In general, there are typing rules that dictate whether a mapping is well-typed with respect to a schema. We shall visit these typing rules in detail, after we formally define schemas.

The most direct semantics of such a mapping is that of a *constraint* between the source schema and the target schema. In this view, a mapping M defines a set of pairs of instances (I, J) with I over the source and J over the target such that (I, J) satisfies the constraint M (in the standard sense). For every I we call a J such that (I, J) satisfies M a *solution* for I with respect to M [10].

[1]We omit one feature of [14]: grouping conditions, which use Skolem functions and set-valued equalities. This is done for expediency, and it is certainly plausible that our results can be extended.

For our example, note that the where clause contains two types of equality conditions. The last equality condition is a target-target condition reflecting a constraint (present on the target schema), while the other four conditions are target-source conditions reflecting the correspondences between source elements and target elements.

Finally, note that not all fields in the schema are mentioned in the mapping (e.g., the target addr field). Such fields may be required to appear in the resulting instance. Synthesizing null values for such fields as well as for the fields that are mentioned by the mapping but are not constrained by the source (e.g., the two target fields eid) is the role of the data exchange process that implements the mapping. (See [10] for a canonical implementation via the chase that constructs *universal solutions.*)

Mapping expressions may be recursively nested inside the second where clause of another mapping expression. Such nesting, in general, is orthogonal to the nesting of the data. As an example of nested mappings, consider the Clio mapping depicted in Figure 2. The four correspondences in Figure 2 maps the source-side undergraduate information into two several target sets. Clio compiles those correspondence into the following nested mapping:

```
for $u$ in $db$.underGrads.underGrad
$\Rightarrow$ exists $s$ in $targetDB$.Enrollment.Student
    where $s$.sid = $u$.sid $\wedge$ $s$.name = $u$.name $\wedge$
          (for $e$ in $db$.enrolls.enroll
           where $e$.sid = $u$.sid
           $\Rightarrow$ exists $c$ in $s$.Courses.course,
                    $e'$ in $targetDB$.Evaluations.eval
              where $c$.cid = $e$.cid $\wedge$ $e'$.grade = $e$.grade $\wedge$
                    $e'$.eid = $c$.eid)
```

Here, source tuples in the underGrad set are mapped into target tuples in the Student set by the outer mapping. The inner mapping requires that all the associated source enrollment tuples (determined by the join condition e.sid = u.sid) are mapped into corresponding target tuples under course and eval. Note that the inner mapping is a constraint that must be satisfied for every binding of the variables in the outer mapping (i.e., u and s).

2.2 Definition of nested mappings

Formally, nested mappings are defined as follows [15]. A *path* is defined by the grammar $p ::= S \mid x \mid p.l$, where S is a schema root, x is a variable, l is a label (attribute within a record), and $p.l$ denotes record projection. The first element in a path (either a schema root or a variable) is also called the *head* of the path. A nested mapping has the form:

```
$M$ ::= for  $x_1$ in $g_1$, ..., $x_n$ in $g_n$
        where  $C_1$
        $\Rightarrow$
        exists  $y_1$ in $g'_1$, ..., $y_n$ in $g'_n$
        where  $(C_2 \wedge M_1 \wedge \cdots \wedge M_n)$
```

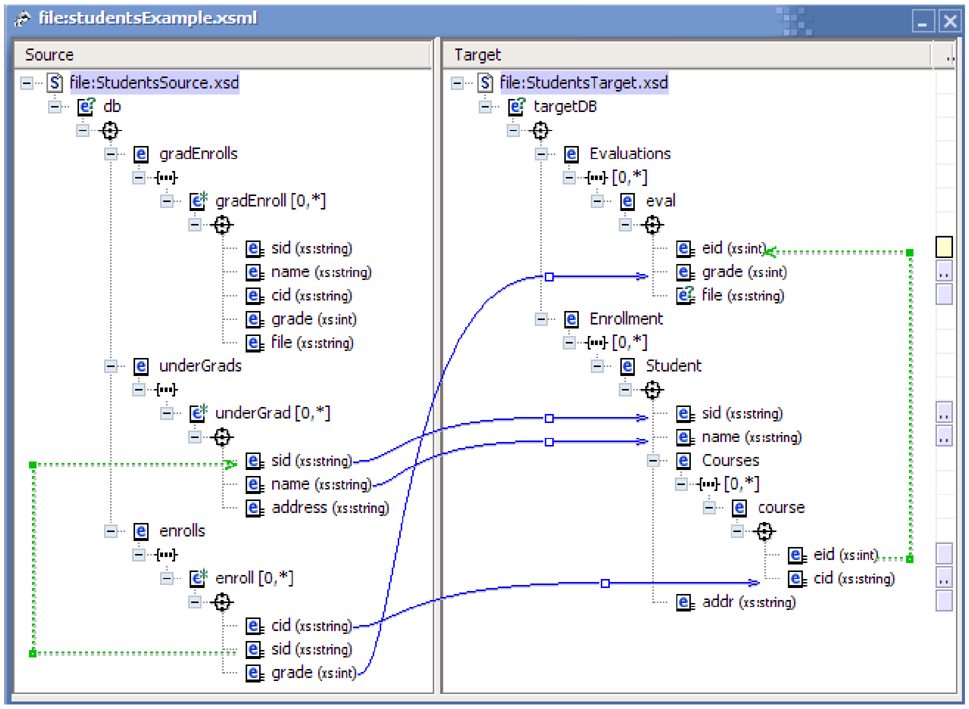

Figure 2: A different mapping

In the above, $M_1, \ldots, M_n$ are *submappings* of M. Each submapping is itself a nested mapping. We say that M is an ancestor of $M_1, \ldots, M_n$ and (recursively) of the submappings of $M_1, \ldots, M_n$. Each x_i <u>in</u> g_i is called a *source generator*. The intention is that the path g_i must evaluate to a set and x_i is bound to an element in that set. Similarly, each y_i <u>in</u> g'_i is called a *target generator*, and the path g'_i must evaluate to a set and y_i is bound to an element in that set. The language supports both sets of records and sets of choice types, which we will describe later.

The following rules must be obeyed for a nested mapping to be well-formed. We define a *source path in* M to be a path whose head is either a source schema root or a variable bound in a source generator of M or bound in a source generator of an ancestor of M. Similarly, a *target path in* M is a path whose head is either a target schema root or a variable bound in a target generator of M or bound in a target generator of an ancestor of M. Then, in a source generator of M, g_i must be a source path in M. In a target generator of M, g'_i must be a target path in M with a similar restriction. The condition C_1 of M consists of a conjunction of equalities between source paths in M. The condition C_2 of M is a conjunction of equalities between target paths in M and equalities between a target path in M and a source path in M. All equalities must be at atomic types.

Even with the help of tools, the full-fledged development of such mappings can be quite complex and can involve significant user effort. Being able to reuse mappings on similar schemas is a crucial feature for real-life metadata applications. In the subsequent part of the paper, we shall explore types and, in particular, polymorphism to show when and how can mappings be reused.

3. NESTED RELATIONAL MODEL

In this section we define the nested relational schema and the nested relational data over which our mappings operate. We define schema and data independently of each other, and then relate the two via a typing relation that can be used to type the data with a schema.

3.1 Nested Relational Schema

The mappings that we consider operate over data instances whose shapes can be described by *nested-relational (NR) schemas* [27]. NR schemas describe atomic types, (unordered) records, (unordered) sets of records, and (unordered) sets of choices (also called variants).

Definition 1 (NR Schema).

$$\begin{array}{rcl} \mathit{Row} & ::= & \llparenthesis\rrparenthesis \mid \llparenthesis\, \mathcal{L} : \mathit{Schema},\ \mathit{Row}\, \rrparenthesis \\ \mathit{Schema} & ::= & \mathtt{ATOMIC}\ \mathcal{A} \mid \mathtt{RCD}\ \mathit{Row} \mid \\ & & \mathtt{SETRCD}\ \mathit{Row} \mid \mathtt{SETCHC}\ \mathit{Row} \end{array}$$

Intuitively, a *Row* is the building block for either a record or a choice. A *Row* is a tuple of *label* : *type* pairs, where *label* is drawn from an infinite set $\mathcal{L}$ of label names, while *type* is one of the four main types under *Schema*. We require that *Rows* contain only one instance of any label name, and we equate *Rows* that are equivalent up to permutation of label name and schema pairs[2]. We shall often abbreviate $\llparenthesis l_1 : t_1, \llparenthesis l_2 : t_2, \llparenthesis\rrparenthesis\rrparenthesis\rrparenthesis$ as $\llparenthesis l_1 : t_1, l_2 : t_2\rrparenthesis$.

`ATOMIC` $\mathcal{A}$ denotes an atomic type, where $\mathcal{A}$ stands for any concrete, non-empty base type (e.g., `Int`, `String`, etc.). For convenience, we will denote `ATOMIC Int` simply as `Int`.

[2]These restrictions are not captured in the syntax, and require special treatment during type inference.

We make several simplifying assumptions in the above definition of complex types. As a result, we depart somewhat from a completely general nested model, although, in practice, we can still capture most XML and relational schemas. Concretely, we do not allow choice values unless they are immediately under a set type. Hence, we explicitly "package" a set of choices into the SETCHC construct. Another restriction is that we disallow sets of sets or sets of atomic types. As a result, we only allow either sets of records or sets of choices, which are packaged as SETRCD or SETCHC. Note, however, that the resulting definition allows RCD, SETRCD, and SETCHC to be freely nested within each other.

We illustrate the above definition with the following two NR schemas, denoted as *src* and *dst* (these names also play the role of the *roots* of the schemas). Here, *src* describes a set of student records, each with a nested set of choices reflecting the teaching/enrollment status of a student, while *dst* describes a set of employee records.

```
src,  RCD ⦇ students : SETRCD ⦇
                   fullname : String,
                   status : SETCHC ⦇ teaching : String,
                                     taking : String ⦈ ⦈ ⦈
dst,  RCD ⦇ employees : SETRCD ⦇ name : String,
                                 job : String,
                                 id : Int ⦈ ⦈
```

For illustration purposes, we give below the corresponding DTD representation for students and employees, where we ignore the fact that DTDs assume ordered sequences.

```
<!ELEMENT students (fullname , status)*>
<!ELEMENT status   (teaching | taking)*> ...
<!ELEMENT employees (name, job, id)*> ...
```

3.2 Nested Relational Data

Although there are many ways to represent nested relations we adopt here particular representation that is based on algebraic datatypes. The results of this paper are independent of the way that nested relational data are represented, but the primary advantage to using this representation is that it has both a simple set-theoretic definition and a simple definition using algebraic datatypes. As such, this representation can be used to exchange nested relational data between mapping systems and programming languages.

Definition 2 (Data Instances). *An instance is constructed inductively as one of the following:*

- *an atom (e.g. 1 or "IBM"),*
- *a pair $(l : d)$ of a label l and an instance d*
- *a set $\{d_1, \ldots, d_n\}$ of instances*

To illustrate, the following is an instance, representing intuitively a set of person tuples, where each tuple includes a name and an age.

$$\{\ \{(\mathsf{name} : \mathit{John\ Doe}), (\mathsf{age} : 25)\},\ \{(\mathsf{name} : \mathit{Alice\ May}), (\mathsf{age} : 22)\}\ \}$$

An instance has no type a priori, and in general it could be given multiple types or it may have no type at all.

We next define, inductively, the typing rules that describe how data can be associated with a schema. In effect, these rules define all the valid data instances for each schema construct. We use $\mathcal{B}(A)$ to represent the domain of an atomic type A. Moreover, we use $[\![X]\!]$ to denote the set of all data instances conforming to schema construct X.

Definition 3 (Typing Data).

$$\frac{}{\emptyset \in [\![\mathtt{SETCHC}\ ⦇⦈]\!]}$$

$$\frac{\forall d \in D, d \in \{(l : i) \mid i \in [\![t]\!]\} \cup [\![\mathtt{SETCHC}\ r]\!]}{D \in [\![\mathtt{SETCHC}\ ⦇l : t, r⦈]\!]} \qquad \frac{}{\emptyset \in [\![\mathtt{RCD}\ ⦇⦈]\!]}$$

$$\frac{d \in [\![t]\!] \qquad e \in [\![\mathtt{RCD}\ r]\!]}{\{(l : d)\} \cup e \in [\![\mathtt{RCD}\ ⦇l : t, r⦈]\!]} \qquad \frac{d \in \mathcal{B}(A)}{d \in [\![\mathtt{ATOMIC}\ A]\!]}$$

$$\frac{\forall d \in D, d \in [\![\mathtt{RCD}\ r]\!]}{D \in [\![\mathtt{SETRCD}\ r]\!]}$$

For example, the RCD rule can be informally read as follows: the instance formed by adding a pair $(l : d)$ to e is of type RCD $⦇l : t, r⦈$, if d is an instance of type t and e is an instance of the record defined by r. Similarly, the SETCHC rule says that a set of instances D are of type SETCHC $⦇l : t, r⦈$ if for all instances $d \in D$, d is either the pair $(l : i)$ where i is of type t, or d is a pair in SETCHC r. An implicit assumption in the RCD and SETCHC rules is that the label l does not occur in r (otherwise $⦇l : t, r⦈$ is not a well-formed *Row*). In effect, these two rules specify how to construct sets of "larger" choices (or "larger" records) from "smaller" data instances. The other interesting rule is the SETRCD rule which specifies how to construct sets of records from records.

For our two NR schemas defined earlier, the following are valid nested relational data instances (the first for *src*, and the last for *dst*):

```
src, {(students : {{(fullname : John Doe),
                    (status :{(teaching : CS100),
                              (taking : CS200),
                              (teaching : CS101)})
                   },
                   {(fullname : Mary Jane),
                    (status : {(taking : CS100),
                               (taking : CS200)})}
                  })}
```

```
dst, {(employees : {{(name : John Doe), (job : CS100), (id : 1)}
                    {(name : John Doe), (job : CS101), (id : 2)}
                   })}
```

Notice that a set of pairs could be typed in general as either a SETCHC or RCD. However, if a set of pairs has two or more pairs with the same label, it cannot be a RCD. For example, the set {(teaching : $CS100$), (taking : $CS200$), (teaching : $CS101$)} in the first instance above can be typed only as SETCHC ⦇teaching : String, taking : String⦈.

As a final example, suppose we have $\{a_1, a_2\} = [\![\mathtt{ATOMIC}\ A]\!]$ and $\{b\} = [\![\mathtt{ATOMIC}\ B]\!]$. Then our typing relation gives the following:

- $[\![\texttt{RCD}\ (\!|i : \texttt{ATOMIC}\ B,\ j : \texttt{ATOMIC}\ A|\!)]\!]$ has two instances:
  1. $\{(i : b), (j : a_1)\}$
  2. $\{(i : b), (j : a_2)\}$
- $[\![\texttt{SETRCD}\ (\!|i : \texttt{ATOMIC}\ B,\ j : \texttt{ATOMIC}\ A|\!)]\!]$ has four instances:
  1. $\{\{(i : b), (j : a_1)\}, \{(i : b), (j : a_2)\}\}$
  2. $\{\{(i : b), (j : a_1)\}\}$
  3. $\{\{(i : b), (j : a_2)\}\}$
  4. $\{\}$
- $[\![\texttt{SETCHC}\ (\!|i : \texttt{ATOMIC}\ B,\ j : \texttt{ATOMIC}\ A|\!)]\!]$ has 8 instances:
  1. $\{(i : b), (j : a_1), (j : a_2)\}$
  2. $\{(i : b), (j : a_1)\}$
  3. $\{(i : b), (j : a_2)\}$
  4. $\{(i : b)\}$
  5. $\{(j : a_1), (j : a_2)\}$
  6. $\{(j : a_1)\}$
  7. $\{(j : a_2)\}$
  8. $\{\}$

Connections to Algebraic Datatypes

The above definition of the nested relational data (with choices) differs from those traditionally found in database literature. However, this representation makes it easy to encode nested relational data using the standard algebraic datatypes 0, 1, +, × and a labeled pair construct. This encoding can be used to transfer data between nested relational systems and programming languages like ML and Haskell. We can break out a separate notion of powerset ($\mathcal{P}$) and choice (CHC), which lets us write data in an equivalent way:

$$\begin{aligned}
[\![\texttt{SETCHC}\ r]\!] &= \mathcal{P}([\![\texttt{CHC}\ r]\!]) \\
[\![\texttt{SETRCD}\ r]\!] &= \mathcal{P}([\![\texttt{RCD}\ r]\!]) \\
[\![\texttt{CHC}\ (\!||\!)]\!] &= 0 \\
[\![\texttt{CHC}\ (\!|l : t, r|\!)]\!] &= \{(l : i) \mid i \in [\![t]\!]\} + [\![\texttt{CHC}\ r]\!] \\
[\![\texttt{RCD}(\!||\!)]\!] &= 1 \\
[\![\texttt{RCD}(\!|l : t, r|\!)]\!] &= \{(l : i) \mid i \in [\![t]\!]\} \times [\![\texttt{RCD}\ r]\!]
\end{aligned}$$

4. CORE MAPPINGS

For most of the results in this paper we will not work with nested Clio mappings. Instead, we will study a slightly more general language, which we call the *core* language. The core language is fully compositional and includes all Clio nested mappings. Although more general, the syntax for this language is simpler, more uniform, and better suited for a type-theoretic treatment. On the other hand, the core language includes expressions that are not mappings in the usual sense.[3] The syntax of core mapping expressions is given by the following definition.

[3]In particular, the source or target restrictions that are important from a schema mapping point of view are ignored. For example, Clio mappings require generator lists to be non-empty; core mappings do not. As a result, core mappings can express constraints over *only* the source or target data, whereas Clio mappings are always non-trivial constraints over *both* the source and target data. Clio represents constraints over only the source or target data as mappings where the source and target roots coincide.

Definition 4 (Core Mapping Expressions).

$$\begin{aligned}
Path &::= v \mid Path.l \\
\diamond &::= \underline{\mathsf{for}} \mid \underline{\mathsf{exists}} \\
\oplus &::= \wedge \mid \Rightarrow \\
M &::= \top \mid Path = Path \mid M \oplus M \mid \\
&\quad\ \ \diamond\, v\ \underline{\mathsf{in}}\ Path\,.\ M \mid \\
&\quad\ \ \diamond\, v\ \underline{\mathsf{of}}\ l\ \underline{\mathsf{from}}\ Path\,.\ M
\end{aligned}$$

In the above, $\top$ represents truth (a true proposition). A $Path$ is an expression that navigates inside a record. A $Path$ is always of the form $v.l_1. \dots l_n$, where v is a variable (or one of the schema roots) and $l_1, \dots, l_n$ are labels. Core mapping expressions (M) are then formed via two forms of variables binders (the $\diamond$ distinguishes universal from existential quantification), one used to navigate inside sets of records and the other used to navigate inside sets of choices. Each bound variable can then be used in a nested mapping subexpression, which in turn can be either another binder, or an equality, conjunction, or implication (possibly empty, or $\top$) of other mapping expressions.

For the $\diamond\ v$ IN $Path$ form, $Path$ must resolve into a set of records (SETRCD) and v will bind to records of that set. For the $\diamond\ v$ OF l FROM $Path$ form, $Path$ must resolve into a set of choice elements (SETCHC); this construct automatically selects only labeled pairs of the form $l : d$, and then v will bind to the data values d.

The following is an example of a core mapping m on the src and dst schemas defined in Section 3.1:

$$\begin{array}{ll}
\text{(m)} & \underline{\mathsf{for}}\ s\ \underline{\mathsf{in}}\ src.\mathsf{students}\ . \\
& \quad \underline{\mathsf{for}}\ t\ \underline{\mathsf{of}}\ \mathsf{teaching}\ \underline{\mathsf{from}}\ s.\mathsf{status}\ . \\
& \quad\quad \top \Rightarrow \\
& \quad\quad \underline{\mathsf{exists}}\ e\ \underline{\mathsf{in}}\ dst.\mathsf{employees}\ . \\
& \quad\quad\quad e.\mathsf{name} = s.\mathsf{fullname}\ \wedge\ e.\mathsf{job} = t
\end{array}$$

A mapping expression can range over multiple source schema roots and target schema roots, which will appear as free variables. The association between a root and its schema (type) is captured by a *context* Γ, which in general is a finite map of bindings from variables to schema:

Definition 5 (Context). $\Gamma ::= - \mid (v, Schema)\,;\, \Gamma$

A *mapping* is a core mapping expression together with a context. (In the Clio language, a mapping is a set of mapping expressions that share a context.)

Definition 6 (Mapping). *A (not necessarily well-typed) mapping is an association* (Γ, M) *between a core mapping expression* M *and a context* Γ *that contains exactly the free variables of* M*.*

Satisfaction

A mapping M can be given a meaning as a constraint between a set of source and target data instances, where one instance is associated with each schema root (whether source or target) of M. Formally, we define an *environment* to be an association from schema roots v to data instances I:

Definition 7 (Environment). $\Delta ::= - \mid (v, I); \Delta$

There is a natural correspondence between contexts and environments. We will write $\Delta \in [\![\Gamma]\!]$ to indicate that each binding $(v, t) \in \Gamma$ has a corresponding binding $(v, I) \in \Delta$ such that $I \in [\![t]\!]$.

Satisfaction, written $\Delta \models M$, is a relation between environments Δ and mappings M, and means that the constraints expressed by M are true when interpreted in the structure Δ. Satisfaction is in general "untyped": it is independent of any notion of schema and it may apply to ill-typed mappings and to data that is not an instance of any schema. Much of the utility of type-checking, which we address next, comes from carving out a subset of mapping expressions (the well-typed ones) that are well-behaved (i.e., are satisfiable, see Theorem 1) over data instances (Definition 2) that conform to the nested relational model. Here we give the definition of satisfaction for the core language; importantly, the semantics of a Clio mapping does not change when translated into the core language. To define satisfaction we must also define path projection, where $\Delta \models p \rightsquigarrow I$ means that the path p evaluates to instance I assuming environment Δ.

Definition 8 (Projection).

$$\frac{(v : I) \in \Delta}{\Delta \models v \rightsquigarrow I} \qquad \frac{\Delta \models p \rightsquigarrow I \qquad \{\, I' \mid (l : I') \in I \,\} = \{\, i \,\}}{\Delta \models p.l \rightsquigarrow i}$$

Definition 9 (Satisfaction).

$$\frac{}{\Delta \models \top} \qquad \frac{\Delta \models m_1 \qquad \Delta \models m_2}{\Delta \models m_1 \wedge m_2}$$

$$\frac{\Delta \models m_1 \;\rightarrow\; \Delta \models m_2}{\Delta \models m_1 \Rightarrow m_2} \qquad \frac{\Delta \models p_1 \rightsquigarrow I \qquad \Delta \models p_2 \rightsquigarrow I}{\Delta \models p_1 = p_2}$$

$$\frac{\Delta \models p \rightsquigarrow I \qquad \forall i \in I,\ (v : i); \Delta \models m}{\Delta \models \underline{\mathsf{for}}\ v\ \underline{\mathsf{in}}\ p.\ m}$$

$$\frac{\Delta \models p \rightsquigarrow I \qquad \exists i \in I,\ (v : i); \Delta \models m}{\Delta \models \underline{\mathsf{exists}}\ v\ \underline{\mathsf{in}}\ p.\ m}$$

$$\frac{\Delta \models p \rightsquigarrow I \qquad \forall (l : i) \in I,\ (v : i); \Delta \models m}{\Delta \models \underline{\mathsf{for}}\ v\ \underline{\mathsf{of}}\ l\ \underline{\mathsf{from}}\ p.\ m}$$

$$\frac{\Delta \models p \rightsquigarrow I \qquad \exists (l : i) \in I,\ (v : i); \Delta \models m}{\Delta \models \underline{\mathsf{exists}}\ v\ \underline{\mathsf{of}}\ l\ \underline{\mathsf{from}}\ p.\ m}$$

5. TYPE CHECKING

One of the goals of our type system is to ensure that well-typed mappings are satisfiable. Our typing relation $\vdash$ is between a core mapping and a context. We give the typing relation by means of inductive inference rules. First, we need rules for typing a path, which we indicate with :: and define below.

Definition 10 (Type-checking Paths).

$$\overset{\text{VAR}}{\frac{(v, t) \in \Gamma}{\Gamma \vdash v :: t}} \qquad \overset{\text{RCD-ELIM}}{\frac{\Gamma \vdash p :: \texttt{RCD}\ (\!|l : t, r|\!)}{\Gamma \vdash p.l :: t}}$$

With this in hand, the main type-checking relation is:

Definition 11 (Type-checking).

$$\overset{\text{WF-EQ}}{\frac{\Gamma \vdash p_1 :: \texttt{ATOMIC}\ a \qquad \Gamma \vdash p_2 :: \texttt{ATOMIC}\ a}{\Gamma \vdash p_1 \ = \ p_2}} \qquad \overset{\text{WF-TRUE}}{\frac{}{\Gamma \vdash \top}}$$

$$\overset{\text{WF-ANDIMPL}}{\frac{\Gamma \vdash M_1 \qquad \Gamma \vdash M_2}{\Gamma \vdash M_1 \ \oplus \ M_2}}$$

$$\overset{\text{SETRCD-ELIM}}{\frac{\Gamma \vdash p :: \texttt{SETRCD}\ r \qquad (v, \texttt{RCD}\ r); \Gamma \vdash M}{\Gamma \vdash \diamond\ v\ \underline{\mathsf{in}}\ p.\ M}}$$

$$\overset{\text{SETCHC-ELIM}}{\frac{\Gamma \vdash p :: \texttt{SETCHC}\ (\!|l : t, r|\!) \qquad (v, t); \Gamma \vdash M}{\Gamma \vdash \diamond\ v\ \underline{\mathsf{of}}\ l\ \underline{\mathsf{from}}\ p.\ M}}$$

The typing rules are syntax directed and are easily read bottom up. For instance, the rule WF-EQ says that to check if an equality constraint is well-formed, we must check if each of the paths is well formed and, moreover, that the two paths have the same atomic type. Checking well-formedness of the paths, in turn, requires repeated uses of RCD-ELIM to check if the required projections exist. Both the typing and type-inference rules treat logical conjunction and implication the same (indicated by $\oplus$), and universal and existential quantification the same (indicated by $\diamond$).

The two more complex rules are SETRCD-ELIM and SETCHC-ELIM. For SETRCD-ELIM, to check that $\diamond\ v$ `IN` $p.\ M$ is well-formed with respect to a context Γ, we must perform two things. First, we must verify that p types to `SETRCD` r for some row r. Then we must check, recursively, that M is well-formed in a new context where Γ is extended with the pair $(v, \texttt{RCD}\ r)$. The SETCHC-ELIM rule is somewhat similar, and involves the additional check that the label l must be one of the valid choices in the `SETCHC` type for p. The following theorem shows that the typing relation achieves our goal of satisfiability, for core mappings.

Theorem 1 (Satisfiability). *Suppose* $\Gamma \vdash M$. *Then we can compute an environment* $\Delta \in [\![\Gamma]\!]$ *such that* $\Delta \models M$.

An example mapping that is both ill-typed and unsatisfiable is exists v in t. $v = v.l$, where t is a schema root and l an arbitrary label. We note also that, for this general form of nested mappings, the above theorem cannot be strengthened to say that we can always find target solutions with respect to M when given an *arbitrary* set of instances over the source schema roots. This is in contrast to the simpler language of s-t tgds in [10] which always admit solutions. It is also in contrast to the more restricted class of nested mappings in [14], which include a complex syntactic check to always guarantee the existence of solutions. However, such a syntactic check can always be added as an orthogonal ingredient on top of our typing system. The following example shows why our nested mappings may not always have solutions (but they are always satisfiable).

```
for s in src.students
  ⊤ ⇒
  exists e in dst.employees.
    e.name = s.fullname ∧
    ( for t of teaching from s.status
      ⊤ ⇒
      e.job = t)
```

If we look at our earlier *src* instance in Section 3.2, it is easy to see that we cannot construct a target solution, since we would have to construct an employee whose name is *John Doe* and whose job is equal to both $CS100$ and $CS101$. Nevertheless, the mapping is satisfiable, since we can always pick some other *src* instance (e.g., with one teaching element in the status set) for which there is a solution.

Finally, we note that we can permit more expressive mappings at the cost of weakening the semantic guarantees provided by the type system. For instance, naively adding atomic valued constants results in typeable mappings that contain unsatisfiable constraints (e.g. $1 = 2$). Similarly, we can add CHC types that are not guarded by a set type (i.e., not packaged as SETCHC) to give rise to typeable mappings that are unsatisfiable.

6. POLYMORPHISM

Our typing relation allows for a typable M to have distinct Γ such that $\Gamma \vdash M$. In other words, there can be different schemas for which M is valid. In this section we extend the schema language with type variables to obtain a formalism for expressing the *principal typings* [30] of mapping expressions, where principal typings completely capture the contexts for which a mapping type-checks. A principal typing corresponds intuitively to the minimum amount of structure that is needed by a mapping to type-check.

6.1 Polymorphic Schema

To begin, we extend the schema language to include row variables ρ, schema variables σ, and atomic variables α in place of atomic type names. We collectively call these variables *type variables.*

DEFINITION 12 (POLYMORPHIC SCHEMA).

$$\begin{array}{rcl} \mathit{Row} & ::= & (\!|\,|\!) \mid (\!|\, \mathit{Row},\ \mathcal{L} : \mathit{Schema}\,|\!) \mid \rho \\ \mathit{Schema} & ::= & \mathtt{ATOMIC}\ \alpha \mid \mathtt{RCD}\ \mathit{Row} \mid \\ & & \mathtt{SETRCD}\ \mathit{Row} \mid \mathtt{SETCHC}\ \mathit{Row} \mid \sigma \end{array}$$

We shall often apply substitutions to polymorphic schemas; a substitution ϕ maps type variables to schema constructs of the same sort (i.e., a row variable can be mapped to a *Row*, a schema variable can be mapped to a $Schema$, while an atomic variable can be mapped to a concrete atomic type name or another atomic variable). Polymorphic schemas are not closed under arbitrary substitutions in the sense that row variables cannot be substituted arbitrarily. For instance, in $(\!|\rho, l : t|\!)$, ρ can only range over rows that do not include l. When we write substitutions we assume that the result is well-formed.

6.2 Principal Typings

First, note that our earlier typing rules (in Section 5) may be used without modification with polymorphic schemas. Our notion of principal typing applies *only* to polymorphic schema, however.

DEFINITION 13 (PRINCIPAL TYPING). *Γ is a principal typing for M iff (1) for every substitution ϕ, we have that $\phi\Gamma \vdash M$, and (2) for every Γ' such that $\Gamma' \vdash M$, there is some ϕ such that $\phi\Gamma = \Gamma'$.*

Thus, the contexts for which a mapping M type-checks are exactly those that can be obtained (via substitution) from the principal typings.

The following is a principal typing for the earlier schema mapping m in Section 4. Here, $\rho_1, \ldots, \rho_5$ are distinct row variables and α_1, α_2 are distinct atomic variables.

$$\begin{array}{l} src,\ \mathtt{RCD}\ (\!|\ \rho_1,\ \mathsf{students} : \mathtt{SETRCD}\ (\!|\ \rho_2, \\ \qquad\qquad \mathsf{fullname} : \mathtt{ATOMIC}\ \alpha_1,\ \mathsf{status} : \mathtt{SETCHC}\ (\!|\ \rho_3, \\ \qquad\qquad\qquad \mathsf{teaching} : \mathtt{ATOMIC}\ \alpha_2\ |\!)\ |\!)\ |\!) \\ dst,\ \mathtt{RCD}\ (\!|\ \rho_4,\ \mathsf{employees} : \mathtt{SETRCD}\ (\!|\ \rho_5,\ \mathsf{name} : \mathtt{ATOMIC}\ \alpha_1, \\ \qquad\qquad\qquad \mathsf{job} : \mathtt{ATOMIC}\ \alpha_2\ |\!)\ |\!) \end{array}$$

Note that the taking and id fields are absent, intuitively because they are not mentioned in the mapping expression. However, by applying a substitution that sends ρ_3 to a row containing a taking label, and sends ρ_5 to a row containing an id label, we obtain schemas that correspond to the original schemas *src* and *dst* of Section 3.1.

PROPOSITION 1. *Principal types are unique up to one-to-one renaming of type variables.*

6.3 Type Inference

In this section we give a sound and complete type inference algorithm that computes the principal typing of a core mapping expression, or fails if one does not exist.

The general idea of the inference algorithm is to use iterated unification, extended to account for permutation of rows.

DEFINITION 14 (UNIFICATION). *A substitution ϕ unifies types t_1 and t_2, written $t_1 \overset{\phi}{\sim} t_2$, when $\phi t_1 = \phi t_2$. A unifier ϕ is most general when for any other unifier ψ, there exists a substitution s such that $\psi = s \circ \phi$.*

Intuitively, during inference we need to unify row expressions like $\llparenthesis l_1 : t_1, l_2 : t_2 \rrparenthesis$ and $\llparenthesis l_2 : t_2, l_1 : t_1 \rrparenthesis$ but traditional unification distinguishes these permutations and cannot unify them. A detailed discussion of such an extended unification algorithm can be found in [17]. We give next our unification algorithm as the set of rules in Definition 15. The reflexivity and symmetry of the rules are not shown explicitly in the definition, but nevertheless assumed. The substitution ϕ is synthesized (inductively) by the rules.

Definition 15 (Schema Unification).

$$\text{BIND}\quad \frac{v \notin fv(x)}{v \overset{v\mapsto x}{\sim} x} \qquad\qquad \text{APPLY}\quad \frac{x \overset{\phi}{\sim} x'}{Cx \overset{\phi}{\sim} Cx'}$$

$$\text{ROW}\quad \frac{(l:t) \overset{\phi}{\in} r' \qquad \phi(r) \overset{\psi}{\sim} \phi(r') - l}{\llparenthesis l : t, r \rrparenthesis \overset{\psi\phi}{\sim} r'} \qquad\qquad \text{INVAR}\quad \frac{\rho' \textit{ fresh} \qquad \rho \notin fv(t)}{(l:t) \overset{\rho\mapsto \llparenthesis l:t,\rho' \rrparenthesis}{\in} \rho}$$

$$\text{INHEAD}\quad \frac{t \overset{\phi}{\sim} t'}{(l:t) \overset{\phi}{\in} \llparenthesis l : t', r \rrparenthesis} \qquad\qquad \text{INTAIL}\quad \frac{(l:t) \overset{\phi}{\in} r \qquad l' \neq l}{(l:t) \overset{\phi}{\in} \llparenthesis l' : t', r \rrparenthesis}$$

In the above, BIND and APPLY are typical unification rules. In BIND, for instance, x is a type or row expression, v is a type variable, and the $v \notin fv(x)$ represents the "occurs check" that v must not occur among the free variables of x. If the premise is satisfied, then v is equivalent to x under a substitution that maps v to x. In APPLY, the notation C is used to mean one of `ATOMIC`, `SETRCD`, `RCD`, and `SETCHC`.

The rules ROW, INVAR, INTAIL, and INHEAD are needed for row unification. They define and use an additional *inserter* substitution ϕ, of $(l : t)$ into r, written $(l : t) \in^{\phi} r$, if $(l : \phi(t)) \in \phi(r)$. The $-$ operator removes a label from a row. The notation $\psi\phi$ represents the composition of substitutions (i.e., apply ϕ first and then apply ψ).

Unification may generate row expressions with duplicate labels, and the inference algorithm must explicitly check for this. For brevity we have omitted these checks in the rules.

Proposition 2. *Schema unification produces most general unifiers.*

Based on schema unification, we are now ready to define the type inference algorithm. As with type checking, the rules for type inference are syntax directed. Substitutions are synthesized (returned), and contexts are inherited (passed as arguments when we go up the rules). We use $\Vdash$ to indicate inference. We begin by performing type inference on paths:

Definition 16 (Type Inference for Paths).

$$\text{VAR-INF}\quad \frac{(v,t) \in \Gamma}{\Gamma \Vdash v :: t} \qquad \text{RCD-ELIM-INF}\quad \frac{\phi\Gamma \Vdash p :: t \qquad \texttt{RCD}\ \llparenthesis l : \sigma, \rho \rrparenthesis \overset{\psi}{\sim} t \qquad \sigma, \rho \textit{ fresh}}{\psi\phi\Gamma \Vdash p.l :: \psi\sigma}$$

We explain the second, more complex rule. The input is an initial context Γ and an expression $p.l$. We first infer that p has type t (under some substitution ϕ). We then verify that t can be written equivalently (via some other substitution ψ) as `RCD` $\llparenthesis l : \sigma, \rho \rrparenthesis$, for some new type variables σ and ρ. Here we use the earlier unification algorithm. If this verification succeeds, then $p.l$ has type $\psi\sigma$, under a new context obtained from Γ by applying both substitutions ψ and ϕ. Note that in this rule all we need to infer about the type of p is that it is a record that contains the label l.

The complete inference algorithm for core mapping expressions is given by the following set of rules.

Definition 17 (Type Inference for Mapping Exps).

$$\text{WF-EQ-INF}\quad \frac{\begin{array}{cc}\phi_1\Gamma \Vdash p_1 :: t_1 & t_1 \overset{\phi_2}{\sim} \texttt{ATOMIC}\ \alpha \\ \phi_3\phi_2\phi_1\Gamma \Vdash p_2 :: t_2 & \texttt{ATOMIC}\ \phi_3\phi_2\alpha \overset{\phi_4}{\sim} t_2\end{array}}{\phi_4\phi_3\phi_2\phi_1\Gamma \Vdash p_1 \ = \ p_2} \qquad \text{WF-TRUE-INF}\quad \frac{}{\Gamma \Vdash \top}$$

$$\text{WF-ANDIMPL-INF}\quad \frac{\phi_1\Gamma \Vdash M_1 \qquad \phi_2\phi_1\Gamma \Vdash M_2}{\phi_2\phi_1\Gamma \Vdash M_1 \ \oplus \ M_2}$$

$$\text{SETRCD-ELIM-INF}\quad \frac{\begin{array}{c}\phi_1\Gamma \Vdash p :: t \\ \texttt{SETRCD}\ \rho \overset{\phi_2}{\sim} t \qquad \rho \textit{ fresh} \qquad \phi_3\phi_2((v, \texttt{RCD}\ \rho); \phi_1\Gamma) \Vdash M\end{array}}{\phi_3\phi_2\phi_1\Gamma \Vdash \diamond\ v\ \underline{\mathsf{in}}\ p.\ M}$$

$$\text{SETCHC-ELIM-INF}\quad \frac{\begin{array}{cc}\phi_1\Gamma \Vdash p :: t & \texttt{SETCHC}\ \llparenthesis l : \sigma, \rho \rrparenthesis \overset{\phi_2}{\sim} t \\ \sigma, \rho \textit{ fresh} & \phi_3\phi_2((v, \sigma); \phi_1\Gamma) \Vdash M\end{array}}{\phi_3\phi_2\phi_1\Gamma \Vdash \diamond\ v\ \underline{\mathsf{of}}\ l\ \underline{\mathsf{from}}\ p.\ M}$$

To give an idea of how the rules work, consider the SETRCD-ELIM-INF rule. We are given an initial (partially inferred) context Γ and a core mapping expression $\diamond\ v\ \underline{\mathsf{in}}\ p.\ M$. First, we infer the type t for p (under some substitution ϕ_1). We then check that this type t unifies with a `SETRCD` ρ type, for some new row variable ρ (and under another substitution ϕ_2). We then extend the context $\phi_1\Gamma$ with a new pair that binds v to `RCD` ρ (under the substitution ϕ_2). We then pass the new context and the mapping expression m to a recursive call to the type inference algorithm. In return, we obtain a new substitution ϕ_3.

In practice, having this algorithm means that given M, we can compute a principal typing Γ, or fail exactly when M is not typable. The following two main theorems of this section capture this precisely. We write $\phi_1 =_{\Gamma} \phi_2$ to indicate that the substitutions ϕ_1 and ϕ_2 are equivalent over the type variables in Γ.

Theorem 2 (Soundness). *For all $\varphi\Gamma \Vdash M$, $\varphi\Gamma \vdash M$.*

Theorem 3 (Completeness). *For all $\varphi\Gamma \vdash M$, there exists S and s such that $S\Gamma \Vdash M$ and $\varphi =_{\Gamma} s \circ S$.*

These properties and the inference algorithm extend straightforwardly to additional operations that have types describable using the system of qualified types in [17]. For instance, atomic constants and function symbols are easy to add, and so is an "erase present field l" operation. (Given a record with fields $(\!|l : t, r|\!)$, the "erase present field l" operation would return a record with fields r). However, an "erase field l if it is present, otherwise do nothing" operation cannot be added as it cannot be typed in the language of [17].

One motivation for using this particular type inference algorithm (and this particular choice for the row unification algorithm, which is in the spirit of the algorithm in [17]) is that the resulting system is compatible with modern functional programming languages like Haskell [31]. This compatibility means that mapping expressions embedded in languages like Haskell may have their principal typings inferred "for free" – surely a win for mapping reusability.

7. POLYMORPHISM AND SEMANTICS

In this section we investigate the meaning of polymorphism and give a semantic notion of principal typing. We begin by making precise the notion that schema structure can be unnecessary for a mapping, and show how instances can have corresponding unnecessary data removed in a satisfiability preserving way. We then give a condition that guarantees that a mapping semantics is indifferent to unnecessary structure and data. We conclude by showing, on a concrete example, how these results can be applied to obtain mapping reuse.

7.1 Subtyping

To structurally compare schemas we define a subtyping relation $(\leq, \preceq)$ as the reflexive transitive closure of the following:

Definition 18 (NR Schema Subtyping).

$$\text{WIDTH}\ \frac{}{(\!|l : t, r|\!) \prec r} \qquad \text{DEPTH}\ \frac{t' < t \qquad r' \preceq r}{(\!|l : t', r'|\!) \prec (\!|l : t, r|\!)}$$

$$\text{SUB-SETCHC}\ \frac{r' \prec r}{\texttt{SETCHC}\ r' < \texttt{SETCHC}\ r} \qquad \text{SUB-SETREC}\ \frac{r' \prec r}{\texttt{SETRCD}\ r' < \texttt{SETRCD}\ r}$$

$$\text{SUB-REC}\ \frac{r' \prec r}{\texttt{RCD}\ r' < \texttt{RCD}\ r}$$

Since the NR schema definition is mutually inductive (it defines both *Row* and *Schema*), our definition of subtyping contains both rules for rows $(\preceq)$ and for schema $(\leq)$. This definition can be used with concrete NR schema, or polymorphic NR schema. Subtyping lifts to contexts and mappings pointwise and respects typability:

Theorem 4 (Mapping Subtyping). *$\Gamma \vdash M$ and $\Gamma' \leq \Gamma$ implies $\Gamma' \vdash M$.*

7.2 Erasure

We might hope that $X \leq Y$ implies $[\![X]\!] \subseteq [\![Y]\!]$. However, this fails for our schema semantics: for example, $\texttt{RCD}\ (\!|l : \texttt{Int}|\!) \leq \texttt{RCD}\ (\!||\!)$, but a data instance of the first record type is never a data instance of the second record type. In the extreme, and as is customary in programming languages, we could achieve this property by changing the schema semantics to be more inclusive (so that, for instance, $\{(l : 0)\} \in [\![\texttt{RCD}\ (\!||\!)]\!]$: define $[\![X]\!]_{\leq} = \bigcup_{X' \leq X} [\![X']\!]$). From a mapping perspective this is a radical and unintuitive departure from the nested relational model, so we will instead relate subtyping to a different, more intuitive, semantic operation of erasure. Principal typings then correspond to spaces of instances that are "maximally" erased.

If Γ and Γ' are two contexts such that $\Gamma' \leq \Gamma$, we can define an operation $erase(\Gamma' \leq \Gamma) : [\![\Gamma']\!] \to [\![\Gamma]\!]$ over a derivation of $\Gamma' \leq \Gamma$ that removes data from the instances in $[\![\Gamma']\!]$ so that they become instances in $[\![\Gamma]\!]$. The definition of *erase* in Figure 3 applies pointwise to the instances in Γ'.

The *erase* operation (and auxiliary operation *erase'*) is similar to a projection operator applied from a subtype to a supertype. As a very simple example, $erase(\texttt{SETRCD}\ (\!|A : t_1, B : t_2|\!) \leq \texttt{SETRCD}\ (\!|A : t_1|\!))$ has the same effect as the standard relational algebra projection of a relation with the A, B attributes to a relation with just the A attribute. But *erase* is more general – in programming language parlance, it is a *subtyping coercion.*

Erasure is directed by the derivation of $X' \leq X$, and in general there may be distinct derivations. For instance, *erase'* is non-deterministic because the first two *erase'* rules may both apply to a given $r' \preceq r$. However, the following proposition holds.

Proposition 3. *For any two derivations a, b of $X' \leq X$, for any $I \in X'$, $erase(a)(I) = erase(b)(I)$.*

For this reason, we will treat *erase* as being a function parameterized by types and not by subtyping derivations.

The following theorem, which is important for mapping reuse, states that *erase* removes unnecessary data in a way that preserve the semantics of a mapping.

Theorem 5. *Suppose $\Gamma \vdash M$ and $\Gamma' \leq \Gamma$ and $\Delta' \in [\![\Gamma']\!]$. Then $\Delta' \models M$ if and only if $erase(\Gamma' \leq \Gamma)\ (\Delta') \models M$.*

As an example, consider the context Γ' given by the *src* and *dst* schemas in Section 3.1. This context is a subtype of the following context Γ:

$$\begin{array}{ll} src, & \texttt{RCD}\ (\!|\,\mathsf{students} : \texttt{SETRCD}\ (\!|\,\mathsf{fullname} : \mathsf{String}, \\ & \qquad\qquad\qquad\qquad \mathsf{status} : \texttt{SETCHC}\ (\!|\,\mathsf{teaching} : \mathsf{String}\,|\!)\,|\!)\,|\!) \\ dst, & \texttt{RCD}\ (\!|\,\mathsf{employees} : \texttt{SETRCD}\ (\!|\,\mathsf{name} : \mathsf{String}, \mathsf{job} : \mathsf{String}\,|\!)\,|\!) \end{array}$$

Furthermore, consider now the *src* and *dst* instances given in Section 3.2. These instances form an environment $\Delta' \in [\![\Gamma']\!]$, which, moreover, satisfies the mapping m given in Section 4 (i.e., $\Delta' \models m$). If we apply $erase(\Gamma' \leq \Gamma)\ (\Delta')$, we obtain the following pair of instances:

Definition 19 (Erasure).

$$\begin{array}{ll}
erase'((\!|l : t', r|\!) \preceq r)(x) & = \{ (l' : v) \mid (l' : v) \in x \wedge l' \neq l \} \\
erase'((\!|l : t', r'|\!) \preceq (\!|l : t, r|\!))(x) & = \{ (l' : erase(t' \leq t)(y)) \mid (l' : y) \in x \wedge l' = l \} \cup \\
 & \quad erase(r' \preceq r)(\{ (l' : y) \mid (l' : y) \in x \wedge l' \neq l \}) \\
erase'(r' \preceq z \preceq r)(x) & = erase(z \preceq r)(erase(r' \preceq z)(x)) \\
erase'(r = r)(x) & = x \\
erase\ (\mathtt{RCD}\, r' \leq \mathtt{RCD}\, r)(x) & = erase'(r' \preceq r)(x) \\
erase\ (\mathtt{SETCHC}\, r' \leq \mathtt{SETCHC}\, r)(x) & = erase'(r' \preceq r)(x) \\
erase\ (\mathtt{SETRCD}\, r' \leq \mathtt{SETRCD}\, r)(x) & = \{ erase(\mathtt{RCD}\, r' \leq \mathtt{RCD}\, r)(y) \mid y \in x \} \\
erase\ (t' \leq z \leq t)(x) & = erase(z \leq t)(erase(t' \leq z)(x)) \\
erase\ (t = t)(x) & = x
\end{array}$$

Figure 3: Erasure

$$\begin{array}{l}
src, \{(\mathsf{students} : \{\{(\mathsf{fullname} : \textit{John Doe}), \\
\qquad\qquad (\mathsf{status} : \{(\mathsf{teaching} : \textit{CS100}), \\
\qquad\qquad\qquad (\mathsf{teaching} : \textit{CS101})\})\}, \\
\qquad\qquad \{(\mathsf{fullname} : \textit{Mary Jane}), (\mathsf{status} : \{\})\}\})\} \\
dst, \{(\mathsf{employees} : \{\{(\mathsf{name} : \textit{John Doe}), (\mathsf{job} : \textit{CS100})\}, \\
\qquad\qquad \{(\mathsf{name} : \textit{John Doe}), (\mathsf{job} : \textit{CS101})\}\})\}
\end{array}$$

It is then immediate to see that $erase(\Gamma' \leq \Gamma)\ (\Delta') \models m$. In other words the satisfaction of m is preserved when we move between instances of Γ' and instances of Γ via *erase*. Of course, the intuition behind this preservation is that m does not use any of the "extra" fields in Γ' (i.e., the fields taking and id).

Principal typings are defined using polymorphic schemas, but we do not have a semantics for polymorphic schemas. However, we can *concretize* principal typings by using a canonical substitution to remove row, schema, and atomic variables. This canonical substitution takes row variables to the empty row, atomic variables to arbitrary (distinct) atomic types, and schema variables to the empty record. Principal typings that have been concretized in such a way denote spaces of instances that cannot be further erased (while still preserving the semantics of mappings). For instance, Γ above is a concretized principal typing of m. A mapping will not type-check with respect to any schema that has less structure than a concretized principal type, and erasure may not be satisfiability preserving for supertypes of the concretized principal type.

7.3 Parametricity

We now relate the above notion of preservation of satisfaction that is based on subtyping and erasure with the more general notion of parametricity. In general, for an arbitrary semantics, we have no guarantees that whenever $\Gamma \vdash M$ and $\Gamma' \vdash M$ the meaning of the mapping (Γ, M) is related to the meaning of the mapping (Γ', M). In contrast, with a parametric semantics (defined below), the meaning of a mapping depends only on the mapping expression, and not on the context with respect to which it type-checks. We show in this section that such a parametric semantics for mappings does exist.

Definition 20 (Parametricity). *A mapping meaning function* $[\![]\!]$ *is parametric if* $\Gamma \vdash M$ *and* $\Gamma' \vdash M$ *imply* $[\![(\Gamma', M)]\!] = [\![(\Gamma, M)]\!]$

In general, not all semantics are parametric. Taking the meaning of a mapping to be the query that a system like Clio generates to implement the mapping (see [19]) usually results in a non-parametric semantics, because fields that do not appear in the mapping expressions may still need to be mentioned in the query (typically, the query must explicitly set these fields to NULL). Likewise, a standard satisfaction-based semantics,

$$[\![(\Gamma, M)]\!] = \{ \Delta \mid \Delta \in [\![\Gamma]\!] \ \wedge\ \Delta \models M \}$$

is not parametric because as the schemas in Γ vary, so do the spaces of instances. However, concretized principal types $\hat{\Gamma}$ are unique, so we can give a parametric semantics by taking:

$$[\![M]\!] = \{ \Delta \mid \Delta \in [\![\hat{\Gamma}]\!] \ \wedge \Delta \models M \}$$

which, by our earlier Theorem 5, is equivalent to a semantics of erased solutions:

$$[\![M]\!] = \bigcup_{\Gamma \leq \hat{\Gamma}} \{ erase(\Gamma \leq \hat{\Gamma})(\Delta) \mid \Delta \in [\![\Gamma]\!] \ \wedge\ \Delta \models M \}$$

(It is immediate that the above semantics is parametric, because no matter the choice of Γ for which $\Gamma \vdash M$, the meaning of M is given in terms of $\hat{\Gamma}$ and therefore invariant.)

A parametric semantics such as above allows a mapping to be applied at different schemas (as long as they are subtypes of a concretized principal typing) with the same meaning. A possible scenario for mapping reuse is one in which a developer creates a mapping M using context Γ. Then the system infers a tighter schema, namely, $\hat{\Gamma}$, which gets rid of all the unused parts. Then M can be automatically applied to other contexts $\Gamma \leq \hat{\Gamma}$. Furthermore, the meaning is the same: M will be satisfied by a set of instances that is the same modulo erasure (i.e., the set of erased instances with respect to $\hat{\Gamma}$ is the same).

7.4 Mapping Reuse

We conclude by returning to the running example from 3.1 to demonstrate, from end-to-end, how the previous results can be used to obtain mapping reuse in real systems like Clio. Before we begin, we remark that one simple but useful way that practical systems like Clio can extend our results here is by allowing the user to specify a "dictionary" of corresponding labels ("synonyms"), so that, for instance, the labels DateOfBirth and BirthDate are considered equal.

Our scenario begins with the user looking at the following source (src) and target (dst) schemas in the Clio GUI:

$$\begin{array}{lll}
src, & \mathtt{RCD}\ (\!|\ \mathsf{students} : \mathtt{SETRCD}\ (\!| & \\
 & \quad \mathsf{fullname} : \mathsf{String}, & \\
 & \quad \mathsf{status} : \mathtt{SETCHC}\ (\!| & \mathsf{teaching} : \mathsf{String}, \\
 & & \mathsf{taking} : \mathsf{String}\ |\!)\ |\!)\ |\!) \\
dst, & \mathtt{RCD}\ (\!|\ \mathsf{employees} : \mathtt{SETRCD}\ (\!| & \mathsf{name} : \mathsf{String}, \\
 & & \mathsf{job} : \mathsf{String}, \\
 & & \mathsf{id} : \mathsf{Int}\ |\!)\ |\!)
\end{array}$$

The user draws lines between the schemas, which are compiled by Clio to the following mapping m:

$$\begin{array}{ll}
(\mathrm{m}) & \underline{\mathsf{for}}\ s\ \underline{\mathsf{in}}\ src.\mathsf{students}\ . \\
 & \quad \underline{\mathsf{for}}\ t\ \underline{\mathsf{of}}\ \mathsf{teaching}\ \underline{\mathsf{from}}\ s.\mathsf{status}\ . \\
 & \qquad \top \Rightarrow \\
 & \qquad \underline{\mathsf{exists}}\ e\ \underline{\mathsf{in}}\ dst.\mathsf{employees}\ . \\
 & \qquad\quad e.\mathsf{name} = s.\mathsf{fullname}\ \wedge\ e.\mathsf{job} = t
\end{array}$$

This mapping expression m, but not the schemas src and dst, is stored by Clio for later use. The mapping expression m has the following principal typing:

$$\begin{array}{ll}
src, & \mathtt{RCD}\ (\!|\ \rho_1,\ \mathsf{students} : \mathtt{SETRCD}\ (\!|\ \rho_2, \\
 & \qquad \mathsf{fullname} : \mathtt{ATOMIC}\ \alpha_1,\ \mathsf{status} : \mathtt{SETCHC}\ (\!|\ \rho_3, \\
 & \qquad\quad \mathsf{teaching} : \mathtt{ATOMIC}\ \alpha_2\ |\!)\ |\!)\ |\!) \\
dst, & \mathtt{RCD}\ (\!|\ \rho_4,\ \mathsf{employees} : \mathtt{SETRCD}\ (\!|\ \rho_5,\ \mathsf{name} : \mathtt{ATOMIC}\ \alpha_1, \\
 & \qquad\qquad\qquad \mathsf{job} : \mathtt{ATOMIC}\ \alpha_2\ |\!)\ |\!)
\end{array}$$

Our scenario continues when the user would like to apply or reuse the previous mapping on the following two schemas, which are modified versions of the earlier schemas:

$$\begin{array}{lll}
src, & \mathtt{RCD}\ (\!|\ \mathsf{students} : \mathtt{SETRCD}\ (\!| & \\
 & \quad \mathsf{fullname} : \mathsf{String}, & \\
 & \quad \mathsf{major} : \mathsf{String}, & \\
 & \quad \mathsf{status} : \mathtt{SETCHC}\ (\!| & \mathsf{teaching} : \mathsf{String}, \\
 & & \mathsf{enrolled} : \mathsf{String}\ |\!)\ |\!)\ |\!) \\
dst, & \mathtt{RCD}\ (\!|\ \mathsf{employees} : \mathtt{SETRCD}\ (\!| & \mathsf{name} : \mathsf{String}, \\
 & & \mathsf{job} : \mathsf{String}, \\
 & & \mathsf{id} : \mathsf{Int}\ |\!)\ |\!) \\
 & & \mathsf{salary} : \mathsf{Int}\ |\!)\ |\!)
\end{array}$$

Since the new schemas are concretizations of the principal typing of m, it follows that the mapping m applies without any changes, with the same erasure semantics as in the original scenario.

8. RELATED WORK

The idea of reusing parts of mappings to construct other mappings has appeared sparingly in the data exchange literature. The work in [22] describes how to use previously computed correspondences between schema elements (a.k.a. *schema matchings*) to enhance the discovery of new schema matchings. In our work, we try to match and reuse *entire* schema mappings that encode source and target schema constraints and a (potentially large) number of correspondences. More recently, [26] explores mapping reuse as part of their *schema exchange* framework. In that work, mappings are expressed between meta-schema models called *schema templates*. Given a mapping between two schema templates and a source schema that is an "instance" of the source schema template, their framework computes a new target schema that is an "instance" of the target template and derives a schema mapping between them. Our work does not depend on creating or maintaining these higher-level mappings between templates and we assume the target schema is given, not created as part of the data exchange process. Further, [26] only reports on relational schemas while our work considers nested-relational schemas.

When mappings can be composed as described in [11, 3], a different reuse strategy is possible. Given a mapping from schema A to schema B and another mapping from schema B to schema C, mapping composition creates (under the correct conditions) a mapping from A to C. In effect, we are "reusing" two existing mappings to create a new mapping between A and C. Our work can be used in this scenario to help find the existing intermediate mappings. For instance, if we are trying to create a mapping between schemas A and C and we already have a mapping from A to B, we can use the techniques in this paper to find and coerce an existing mapping into a mapping between B and C.

Finally, we note that nested relational data can be represented using trees, and the programming languages community has a wealth of knowledge about transformations on tree-like data [12], including bi-directional tree transformations [18]. The XML processing languages and systems XDuce [20] and Xtatic [16] aim to create general XML processing languages where XML values are first-class. The languages are functional in nature and have an intuitive semantics for XML processing. They introduce a rich language of types to describe XML values (including regular expressions). The specificity of types for XML, however, leads to restrictions on polymorphism, function types, and type inference that have only recently been addressed [29]. These systems are, in a certain sense, the XML counterparts of LINQ [24]. There is also a fair amount of work on schema inference for SQL and relational algebra (e.g. [28, 7]).

9. CONCLUSION AND FUTURE WORK

We have implemented our ideas as an extension to Clio. The extension is able to infer types of mappings, reuse mappings at different schema, and can automatically populate mapping graphs through schema-analysis. The extension also has experimental support for features not described in this paper, including reuse in the presence of target-side foreign key constraints, the ability to rewrite a mapping from a schema to apply it to a larger (in a schema containment sense) schema, and support for a mapping language extension that is able to express mappings that depend on other mappings [31]. The latter feature has applications, for instance, in dataflow graphs of mappings when a mapping downstream depends on a mapping upstream.

As a future direction, we are studying how to support recursive schema languages. Some schema languages used in data exchange allow recursion (e.g., XML schemas) but the mapping and schema languages defined in this paper do not have syntax for recursion. Clio deals with recursive XML schema by unfolding them a fixed number of times while translating them into NR schema. We are working to extend the mapping and NR schema languages to handle recursion and investigating reuse with the resulting languages.

10. REFERENCES

[1] M. Arenas, J. Pérez, and C. Riveros. The Recovery of a Schema Mapping: Bringing Exchanged Data Back. In *PODS*, pages 13–22, 2008.

[2] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *CIDR*, pages 209–220, 2003.

[3] P. A. Bernstein, T. J. Green, S. Melnik, and A. Nash. Implementing Mapping Composition. In *VLDB*, pages 55–66, 2006.

[4] P. A. Bernstein and S. Melnik. Model Management 2.0: Manipulating Richer Mappings. In *SIGMOD*, pages 1–12, 2007.

[5] P. A. Bernstein, S. Melnik, and J. E. Churchill. Incremental Schema Matching. In *VLDB (demo)*, pages 1167–1170, 2006.

[6] V. Borkar, M. Carey, D. Engovatov, D. Lychagin, T. Westmann, and W. Wong. XQSE: An XQuery Scripting Extension for the AquaLogic Data Services Platform. In *ICDE*, pages 1307–1316, 2008.

[7] J. V. den Bussche, D. V. Gucht, and S. Vansummeren. A crash course on database queries. In *PODS*, pages 143–154, 2007.

[8] R. Fagin. Inverting schema mappings. *ACM TODS*, 32(4), 2007.

[9] R. Fagin, L. M. Haas, M. A. Hernández, R. J. Miller, L. Popa, and Y. Velegrakis. Clio: Schema Mapping Creation and Data Exchange. In *Conceptual Modeling: Foundations and Applications, Essays in Honor of John Mylopoulos*, pages 198–236. Springer, 2009.

[10] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[11] R. Fagin, P. G. Kolaitis, L. Popa, and W. Tan. Composing Schema Mappings: Second-Order Dependencies to the Rescue. *TODS*, 30(4):994–1055, 2005.

[12] J. N. Foster, B. C. Pierce, and A. Schmitt. A logic your typechecker can count on: Unordered tree types in practice. In *PLAN-X, informal proceedings*, Jan. 2007.

[13] M. Friedman, A. Y. Levy, and T. D. Millstein. Navigational Plans For Data Integration. In *AAAI/IAAI*, pages 67–73, 1999.

[14] A. Fuxman, M. A. Hernández, H. Ho, R. J. Miller, P. Papotti, and L. Popa. Nested Mappings: Schema Mapping Reloaded. In *VLDB*, pages 67–78, 2006.

[15] A. Fuxman, M. A. Hernandez, H. Ho, R. J. Miller, P. Papotti, and L. Popa. Nested mappings: schema mapping reloaded. Technical Report CSRG-561, Department of Computer Science, University of Toronto, 2007.

[16] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. The Xtatic experience. In *PLAN-X*, Jan. 2005. University of Pennsylvania Technical Report MS-CIS-04-24, Oct 2004.

[17] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, November 1996.

[18] M. Greenwald, J. Moore, B. Pierce, and A. Schmitt. A language for bi-directional tree transformations. Technical report, Department of Computer and Information Science, University of Pennsylvania., 2003.

[19] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio Grows Up: From Research Prototype to Industrial Tool. In *SIGMOD*, pages 805–810, 2005.

[20] H. Hosoya and B. C. Pierce. Xduce: A statically typed xml processing language. *ACM Trans. Inter. Tech.*, 3(2):117–148, 2003.

[21] M. Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, pages 233–246, 2002.

[22] J. Madhavan, P. A. Bernstein, A. Doan, and A. Y. Halevy. Corpus-based Schema Matching. In *ICDE*, pages 57–68, 2005.

[23] J. Madhavan and A. Y. Halevy. Composing Mappings Among Data Sources. In *VLDB*, pages 572–583, 2003.

[24] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD*, page 706, 2006.

[25] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema Mapping as Query Discovery. In *VLDB*, pages 77–88, 2000.

[26] P. Papotti and R. Torlone. Schema exchange: Generic mappings for transforming data and metadata. *Data Knowl. Eng.*, 68(7):665–682, 2009.

[27] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating Web Data. In *VLDB*, pages 598–609, 2002.

[28] J. van den Bussche and E. Waller. Type inference in the polymorphic relational algebra. In *PODS*, pages 80–90, 1999.

[29] J. Vouillon. Polymorphic regular tree types and patterns. In *POPL 06*, pages 103–114, New York, NY, USA, 2006. ACM.

[30] J. B. Wells. The essence of principal typings. In *ICALP '02*, pages 913–925, London, UK, 2002. Springer-Verlag.

[31] R. Wisnesky. Mapping dependence. Technical Report TR-09-09, Harvard University Computer Science Group. Available at ftp://ftp.deas.harvard.edu/techreports/tr-09-09.pdf, 2009.

[32] R. Wisnesky, M. A. Hernandez, and L. Popa. Mapping polymorphism - proofs. Technical Report TR-10-09, Harvard University Computer Science Group. Available at ftp://ftp.deas.harvard.edu/techreports/tr-10-09.pdf, 2009.